UNIVERSITY OF GOTHENBURG

GPU PATH TRACING

Master of Science Thesis in the Programme Computer Science

JINCHENG LI

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Göteborg, Sweden, July 2010

GPU Path Tracing

JINCHENG LI

Examiner: Ulf Assarsson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden July 2010

Abstract

The goal of this work is to verify the possibility to utilize GPU for global illumination computations in a commercial software environment and explore an efficient way to do it. Path tracing with BVH as the acceleration data structure was implemented on GPU using *CUDA* successfully. It was arranged as a pipelined structure which supported mulitple texture types and light source types. And it also proved to be efficient in execution and compact in memory.

A light tree structure was introduced in this work as well, which grouped light sources with an affecting radius hierarchically and enabled the path tracer to handle massive light sources efficiently. Testings and analyses were also made for different configurations to utilize the light tree.

Keywords: global illumination, path tracing, *CUDA*, massive light sources

Acknowledgments

First of all I would like to thank my supervisors Ulf Assarsson and Magus Pettersson for their kind guidance though this work. Further appreciation go to Anders Nilsson, Niklas Harrysson and David Larsson. They gave me a lot of useful advices and answered many questions.

I would also like to thank all the staffs in the company Illuminate Labs for all the resources they provided, including the machine, books, game scenes and nice meals. I am very thankful for having the chance to be at Illuminate Labs for these ten months.

Last but not least, I would like to thank my fellow thesis worker Morgan Bengtsson as well. He helped me and also supported me throughout the time at Illuminate Labs.

Table of Contents

1 Introduction

1.1 Background

1.1.1 The Quest for Global Illumination

The quest for visual realism can be traced through the history of art, when people discovered the law of perspective drawing and some heuristics of realistic shading as time went by [1]. During the development of computer science, the topic of how to synthesize a photorealistic image on computers forms a sub-field of computer science which is known as global illumination [1]. Global illumination benefits many industries, one of which is computer games [1].

1.1.2 Global Illumination in Games

By applying global illumination technologies, more visual effects, such as color bleeding and ambient occlusion, can be gained in computer games, which make it look more realistic. Figure 1 shows an example.

Figure 1: The left image is rendered without global illumination. While the right one is rendered using global illumination with max 4 light bounces. The game level comes from Epic Games, Inc..

Full global illumination is so costly in computation that it is rarely rendered in realtime [2]. So certain kind of data is usually precalculated offline and read back from memory during the realtime rendering stage. The game *Quake II* (1997) by id Software first made use of this technology in

commercial applications [2]. The global illumination information was computed and stored in the form of *light maps*, which are textures recording irradiance values [2]. Other techniques based on the idea of precompuation were also developed, such as *directional light maps* [2] and *irradiance volumes* [2], aimed at less memory consumption and character relighting respectively.

Research on realtime approximation of global illumination effects are also hot topics. An achievement in this field is the *screen space ambient occlusion (SSAO)*, which was first pinoeered by Crytek in 2005 and is now used widely in high-quality games [3]. The *reflective shadow maps* is another screen space method that can render plausible indirect illumination efficiently [4]. If indirect shadows are considered, the *imperfect shadow maps*, which is an approximation method for visibility, can be used in conjuction with virtual point lights based methods, such as the *reflective shadow maps*, to achieve better global illumination effects for dynamic scenes in realtime frame rates [5]. Though realtime techniques fit dynamic scenes better, precomputation methods provide images with much higher qualities and cheaper runtime costs and they are still widely used today.

1.1.3 GPU in Global Illumination

The GPU is rich in its computational power and can also be utilized for global illumination computation. Algorithms such as *instant radiosity* was designed specifically for the GPU [2]. Though it aimed to take advantage of fixed-function graphics hardware, it also maps to programmable GPUs very well [2]. Other methods were also developed to solve the global illumination problem on programmable GPUs, for instance, the work by Purcell et al. [6] and Sloan et al. [7].

With the appeareance of the *CUDA* architecture in 2007 [8], general computations on GPUs become more flexible and convenient, and much recent work on GPU solutions to the global illumination problem (which will be overviewed in section 1.3) took advantage of it.

Furthermore, the ray tracing algorithm, which typically is used for high quality global illumination, can be implemented more efficiently on GPUs with the new *Fermi*-architecture [9]. NVIDIA also released a new *ray tracing* software package *Optix* for commercial use. It is reasonable to believe that GPUs will play a more and more important role in solving global illumination problems.

1.2 Problem Statement

As discussed in section 1.1.2, precomputation is still an important way to improve the visual quality of computer games. Combined with the discussion in section 1.1.3, the problem of how to utilize GPUs in offline rendering to make the precomputation of global illumination effects of static scenes efficient becomes more practical and valuable.

The following problem is then brought up: is it possible to make a GPU implementation to compute global illumination effects, which is able to be integrated into a commercial software environment? More precisely, it should be:

1. able to compute global illumination for static scenes with high quality,

2. efficient to execute,
3. economic in memory consumption to be able to fit real game levels,
4. able to support multiple texture types and light source types,
5. able to handle massive amouts of light sources,
6. extensible.

This report answers this problem by presenting and discussing a *CUDA* implementation that fulfills the requirements above.

1.3 Related Work

Among all the global illumination algorithms, *ray tracing* algorithms have been developing for 30 years by now [1] and used very commonly. In this work, the term *ray tracing* refers to a category of algorithms, to which *Whitted-style ray tracing*, *distributed ray tracing*, *path tracing* and *bidirectional path tracing* belong. *Ray tracing* algorithms can partially solve, such as *Whitted-style ray tracing*, or fully solve, such as *path tracing*, the global illumination problem.

There are several works which implemented *ray tracing* algorithms on GPU to solve the global illumination problem. Popov et al. [10] presented a GPU-based, stackless *kd*-tree traversal algorithm for ray packets and they achieved a performance that was similar to its CPU conterpart. However, for the performance of secondary rays, they only demonstrated reflection and refraction rays.

In the work by Günther et al. [11], a more compact acceleration data structure, which was *Bounding Volume Hierarchy* (BVH), taking the place of the *kd*-tree. It was used for ray packet traversal algorithm with a shared stack. Though, in the report of Zhou et al. [12], a *kd*-tree can be built efficiently enough in realtime with high quality, Günther et al. [11] argued that a BVH only needs ⅓- ¼ of the memory of a *kd*-tree and is one order of magnitude smaller than a *kd*-tree with ropes. The compactness property makes BVH a suitable solution for a GPU to fit large scenes. Additionally, Günther et al. [11] also reported that they achieved comparable or even slightly faster speed than a *kd*-tree method. But their discussion was limited to primary and shadow rays.

The works above, [10] and [11], both applied packet traversal techniques, and in [10], a comparision between the single ray traversal and the packet traversal was made for stackless *kd*-trees on the G80 architecture. It showed that the packet traversal outperformed the single ray traversal algorithm. Similar comparisions were also made for CPU algorithms. Boulos et al. [13] reported speed-ups for packet traversal of reflection and refraction rays by exploring the available coherence in a BVH. Månsson et al. [14] researched the coherence inherented for secondary rays for different depth and different heuristics to explore it. While the effect of packet traversal for BVHs on the newer GTX200 architecture was inconsistent with the reports above, as discussed in the work of Aila and Laine [15]. In their report, the packet traversal algorithm never exceeded the single ray traversal counterpart for both primary rays and secondary rays in any scene tested. They also provided testings for diffuse reflection rays to demonstrate the affects of incoherent rays. What is more, they also found a new way to distribute works for threads on a GPU, which brought significant speed-ups.

In addition to *ray tracing* algorithms, *photon mapping* is another popular global illumination algorithm. Wang et al. [16] presented a GPU implementation of *photon mapping* with *final gathering* and even achieved interactive frame rates. The light paths it can consider are very rich so that it is able to capture effects such as caustics which are not so efficient to calculate by most *ray tracing* algorithms. However, it also has drawbacks, such as only supporting low-frequency glossy materials and containing bias in the image.

1.4 Thesis Structure

Chapter 2 introduces some global illumination algorithms with more details and also states the algorithm and scheme used in this work. The implementation details are stated in chapter 3. The results are shown and analyzed in chapter 4. Chapter 5 concludes this work and discusses the future work.

2 Alogrithm Descriptions

2.1 Global Illumination Algorithms

2.1.1 *Radiosity* Algorithms

There are two major categories of global illumination algorithms: *radiosity* algorithms and *ray tracing* algorithms. The most excellent feature of *radiosity* algorithms over *ray tracing* algorithms is that they can compute a world space representation of the illumination as a prepass, which makes the rendering of a new camera view very efficient. But the drawbacks of this kind of algorithms are also obvious. They are *biased* and *inconsistent* methods which means the results may never converge to the correct values. Figure 2 shows an example. In the left image, the shadow within the red line boundary blurs incorrectly at the angle of the cube. While the image on the right shows the correct looking, which is rendered using *path tracing*. This kind of bias comes from the fact that the scenes are discretized into small patches in *radiosity* algorithms and they can fail to capture occlusion properly in this way. Another source of bias is that only diffuse interreflections are counted for which makes the algorithms always neglect certain types of light paths.

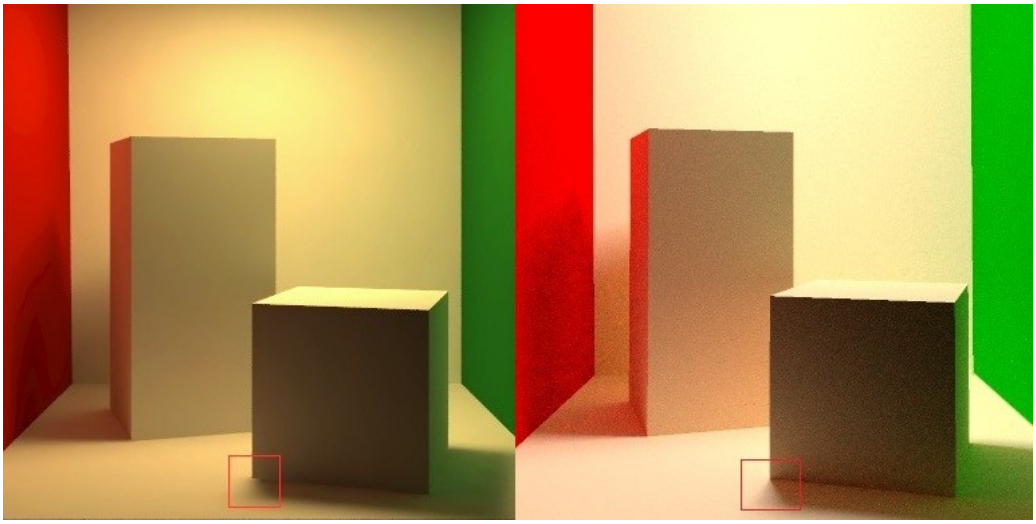


Figure 2: The cornell box on the left is rendered by a radiosity algorithm (Image courtesy of Cornell Box, Cornell Univerity). The right one is rendered by the path tracing algorithm used in this work.

2.1.2 *Photon Mapping*

As mentioned in section 1.3, *photon mapping* is another popular global illumination algorithm. Similar to *radiosity* algorithms, *photon mapping* is also able to generate new camera views efficiently by reusing the *photon map* calculated priorly. Additionaly, it can capture all types of light paths as well. But, for the standard *photon mapping* algorithm, the biases it incurs make it often fail to capture some small details correctly. See figure 3 for an example. By using *final gathering* in the second pass of *photon mapping*, these errors can be alleviated. Theoretically, to make the errors in the image infinitely small, the number of photons needed grows to infinity, which requires the

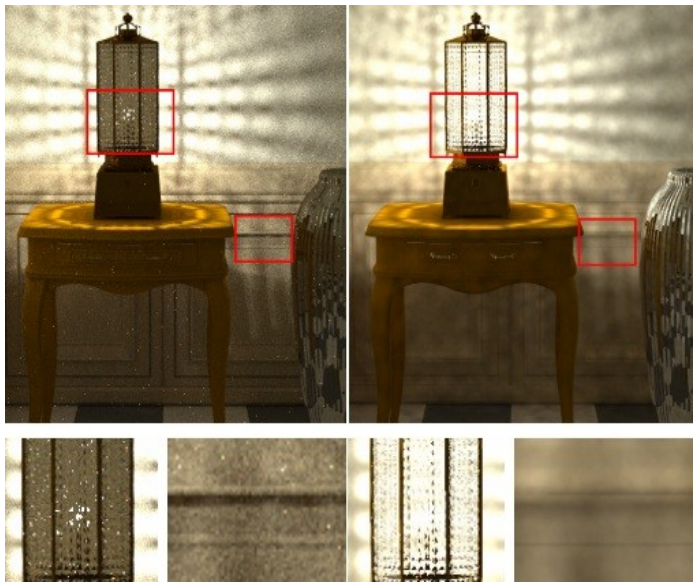


Figure 3: The image on the left is rendered using bidirectional path tracing while the right one is rendered using photon mapping. As can be seen from the pictures, the occlusion in the right image looks very biased. These images come from the work [17].

memory storing the *photon map* to be infinitely large. Hachisuka et al. [17] solved this problem by introducing the *progressive photon mapping* recently, which enables the algorithm to achieve arbitrarily good results with limited amount of memory. This method is good at capturing the specular-diffuse-specular light paths compared to *ray tracing* algorithms. However these paths are not so important in games and often ignored in precomputation methods such as the *light map*. Besides, this new approach discards the data reuse feature of the standard *photon mapping* algorithm and thus everything must be calculated from scratch when a new camera view is rendered.

2.1.3 *Path Tracing*

The *path tracing* algorithm is very simple and able to fully solve the global illumination problem. It is *unbiased* and *consistent*, which means a more correct result can be found from averaging several coarse approximations, so the memory it requires is limited and it is able to get all global illumination effects. Though other approaches, for instance *irradiance caching*, may be more

efficient and outperform the standard *path tracing* on a CPU, the simplicity of *path tracing* makes it suitable to be mapped onto highly-parallel machines, such as CPU clusters or a GPU. Since there is almost no communication overhead, the parallel *path tracing* algorithm is very scalable and can gain the most benefits from a brute-force but easy implementation. All these merits make the *path tracing* algorithm fit the goal of this work the most.

2.2 The Acceleration Data Structure

In the work by Zlatuška and Vlastimil [18], the performances of tracing primary and secondary rays on GPUs with three different accerlation data structures, which were uniform grids, *kd*-trees and BVHs, were compared and analyzed. They reported that uniform grids were only superior for uniformly populated scenes. In the scope of this work, this is rarely the case and makes the uniform grids not a good choice.

As to the comparison between *kd*-trees and BVHs, they concluded that BVHs were better for coherent rays, such as primary rays and shadow rays, while *kd*-trees could be more efficient on average for incoherent rays. But, as mentioned in section 1.3, the memory requirement of a *kd*-tree is much larger than of a BVH. So BVH is used as the spacial acceleration data structure in this work.

2.3 Single Ray Traversal Versus Ray Packet Traversal

Some work, including [13], reported speed-ups for tracing packets of rays for both primary and secondary rays on CPU if packets were managed carefully. This scheme often explores the available coherence among rays in a better way resulting in less redundant memory access, better cache coherence and SIMD fashion for rays in a packet. But the difference between the architecture of CPU and GPU makes some of these advantages pointless. For instance, all the threads within a *warp*[1] [8] are executed in a SIMD fashion natively and there is no cache hierarchy for most of the off-chip memory [8] before the *Fermi* architecture [9].

Two different observations of applying packet traversal techniques on GPUs were reported by [10] and [15]. In [10], tracing packets of rays was more efficient than single ray tracing for both primary and secondary rays, while, in contrast, [15] reported that single ray tracing was always the winner. The most possible reason is the evolution of the GPU's architecture. The testings in [10] were done on the G80 architecture, on which the accesses to the same address within a *half-warp* are serialized [8], and in such cases, packet traversal can greatly save the lantencies and bandwith. On the other hand, the GTX200 architecture was used in [15] and the new feature supporting reading efficiently from the same address makes the above advantage disappear. Another possible explanation is that [10] used a stackless *kd*-tree while [15] used shared stacks for packets in a BVH. Because extra overheads for manipulating shared stacks were incurred, the performance of packet traversal used in [15] was even worse.

In this work, tracing incoherent secondary rays is very imporatant because full global illumination effects are needed. As analyzed in the work [14], as the depth of ray paths grows, the coherence

1 A warp refers to 32 consecutive threads in CUDA which were executed in a SIMD fashion.

among rays drops sharply for most senarios, which may have great negtive impacts on packet traversal methods. Besides, a single ray traversal method is also much easier to implement. Aiming at better utilizing state-of-the-art GPUs, this work traces single rays separately.

2.4 Dealing with Massive Amount of Light Sources

2.4.1 The Light Tree

If an object is not emissive and only lit by a light source, the *radiance* it reflects from that light source degrades to the square of the distance between them.[1] So the affect of a light source to a certain object weakens sharply as one moves farther from the other. Some game engines apply non-physical based light models which provide plausible visual effects. For instance, *UnrealEngine 3* uses such a model for point lights that the light influence degrades to zero if it is beyond an affecting radius. By using this radius, points being shaded can cull important light sources efficiently. Though applying this scheme will incur biases in the image, the influence is not so obvious and the increase in performance is significant.

During the shading process, each shading point needs to find all the light sources that have this point in range. The simplest method is to do a full light loop while the complexity is linear to the number of light sources in the scene. This is not so efficient.

If the affecting radius of each light source is small compared to the size of the whole scene, different light sources can be grouped into a tree structure based on their affecting radius. This light tree structure can be used for light sources with an affecting radius, such as point lights and spot lights. While, for light sources without a radius, for instance, directional light, it can hardly be helpful. An illustration of the light tree can be seen in figure 4.

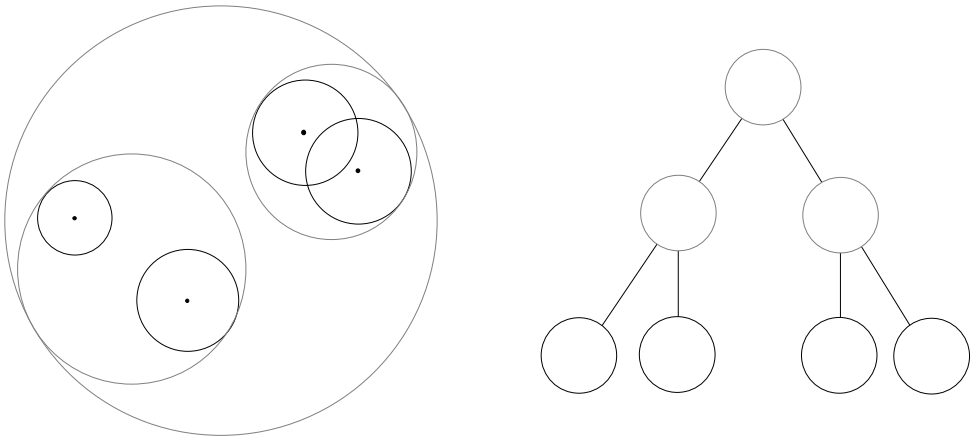


Figure 4: The circle hierarchies on the left shows the light tree structure. The circles in black present different light sources with the positions of each light source. The gray circles group the light sources hierarchically. The tree structure of this hierarchy shows on the right.

1 See appendix A.1.1 and A.1.3.

2.4.2 Light Source Sampling

Based on the theory of *Monte Carlo Methods* [1], instead of looping through all valid light sources, the contribution of multiple light sources to a point can be estimated using sampling schemes. Then the *radiance* coming out from a surface point x can be estimated as:

$$L_{est}(x) = L_{emi}(x) + \frac{1}{N}\sum_{i=1}^{N}\frac{\int_{A_i} f_r(x)\,L_i(y)\,V(x,y)\,G(x,y)\,dA_y}{p_i} \quad .^{[2]}$$

$L_i(y)$ is the outgoing radiance of the ith sampled light source at position y along direction $y \rightarrow x$. The probability to select that sample is p_i. And N is the total number of samples.

The simplest scheme to pick samples is to select them randomly among all the light sources in the scene. The variance generated by this scheme equals

$$\sigma^2 = \frac{1}{N}\left(\int\int \frac{f(x,y)^2}{p(x,y)}dxdy - I^2\right) \quad ,^{[3]}$$

where

$$p(x,y) = p_{pt}(x)\,p_{ls}(y) \quad ,$$

and

$$\int_A p_{ls}(y)\,dy = 1 \quad ,$$

with the integral domain A the entire surface area of all the light sources.

The light tree structure discussed above can be utilized in this sampling process. If the light tree is applied, the probability becomes

$$p(x,y) = p_{pt}(x)\,p_{ls}'(y|x) \quad ,$$

satisfying the equation that

$$\int_{A'} p_{ls}'(y|x)\,dy = 1 \quad ,$$

with the integral domain A' the surface areas of the light sources within range. Obviously, in most cases, A' is smaller than A and it makes the probability $p(x,y)$ higher for a certain point x in the case of light tree sampling, which makes the entire variance smaller.

1 See appendix A.2.
2 See appendix A.1.3.
3 See appendix A.2.

3 GPU Implementations

3.1 The Implementation of BVH

The BVH for the geometries in the scene was built on CPU in this work, using a greedy *Surface Area Heuristic* (SAH) algorithm in a top-down fashion. Specifically, the entire scene was first expressed as a triangle list. Then, by using the splitting scheme described in the work [19], big triangles were split into smaller ones based on a user-defined threshold. Then, the tree was built in the depth-first order. To make a good split for each node, a sampling based SAH approach was applied, which was similar to the one in the work [20]. In this work, 8 samples per axis were used and the split with the minimum SAH cost among the 24 candidates was selected for each step. Then the triangle list was reordered based on this split. When the number of triangles in the current node was below a certain threshold, which was 8 in this work, the node became a leaf.

3.2 The Traversal Kernel and Data Storage

Following the work [15], the dual sibling nodes were tested together in the traversal code and a dynamic working queue was held for each thread warp in this work. What is more, Woop's method [21] was used for ray-triangle intersection test as well, as presented in [15].

As mentioned in the previous section, the entire scene was stored as a triangle list, where each triangle was expressed by a transformation matrix as described in [21]. Instancing was not supported in this work and all the instances in the scene were replaced by seperate groups of triangles in the triangle list. All the other related data, such as the per-vertex normal list and the per-triangle material index list, were also stored without instancing in the same way. The BVH tree was stored as described above, in a 1D texture to benefit from the texture cache. While the other data, including the geometry data and per-ray data, were stored as structure-of-arrays in the non-cached global memory. Additionally, the addresses of these arrays were stored in the constant memory, where each GPU thread can access freely and efficiently, as opposed to passing them by long argument lists, to save the shared memory space.

3.3 The Shading Kernels

Since *CUDA* does not support arrays of texture references, the number of textures that can be fetched from any *CUDA* kernel at a time is fixed. Different game levels often contain different number of textures with different sizes, so a dynamic texture binding scheme was used in this work. Specifically, all the textures were streamed onto GPU memory at the start of the program. Then, after the execution of the traversal code in each pass, the rays were sorted based on the material

indices of their hit positions. A loop over all the active materials in the current pass was then needed. For each active material in the loop, different types of textures were dynamically bound to the texture references. The supported types of textures in this work were diffuse textures, emissive textures and specular textures. While the support for more types of textures could be easily extended.

After the dynamic texture binding, the active number of rays/threads may not be enough for efficient utilization of the hardware. So a seperate material color fetching kernel was made, aiming to reduce the workload for inefficient execution. This kernel read data from the active textures and did a little processing on the data and then stored the intermediate results. Additionally, this kernel also updated each ray's status, including its direcion and origin, based on the material of the hit surface. For a diffuse reflection, *cosine sampling* scheme [1] was used, so the update for the per-ray weight was simplified by riding the cosine term. What is more, the *Phong* model was used for the BRDF in this work. And, for simplicity reasons, the factor *shininess* was always set to 0 which means only mirror reflection was considered for specular reflections.

When the loop above finished, the direct lighting kernel was called. This kernel computed the direct lighting for the hit positions of the rays which had not terminated yet. It shot a user-defined number of shadow rays per hit position and combined the intermediate results from the last kernel to produce the correct shading results. The light sampling schemes discussed in 2.4.2 were used in this kernel.

3.4 The Light Tree Building and Sampling

Similar to the construction of the BVH, the light tree was also built on CPU in a top-down fashion. The greedy scheme was used for spliting nodes in each step, which was that the binary split that resulted in the minimum sum of the two children's volume was always prefered. The reason for applying this simplified scheme insteading of using the standard SAH scheme was that the light tree was used for accelerating locating points rather than shooting rays. Additionally, all the possible binary splits along each axis were evaluated to make an optimal choice because the amount of light sources was much smaller compared to the amount of geometries typically.

The light tree was stored in GPU's global memory after the construction. Different types of light sources were presented using different structures and light sources in the same type were stored as array-of-structures in the global memory as well. There was an identifier indicating the type of light source in each leaf node of the light tree. In this work, a switch block was used to sample light sources of different types[1]. So arbitrarily many types of light sources could be supported by defining more structures and adding more branches in the switch block. Though this scheme resulted in a large and branchy kernel function, the time spent on it was small and it was also easy to implement. An alternative to this will be discussed in section 5.2.

A traversal algorithm without the per-ray stack was used for the light tree. Each ray was tested from the root node and descended towards leaves recursively. If both children nodes were valid for descending, one of them was chosen randomly based on a probability[2]. This scheme made both the

1 The supported light source types in this work were point lights, rectangle area lights and directional lights. Because the directional lights were not included in the light tree, they were handled using another kernel.

2 It was 50 percent in this work.

implementation and the execution easier. Results with less variance could be found by simply shooting more shadow rays.

3.5 The CPU Host

As shown above, the entire process of calculating global illumination effects was executed on GPU and split into small kernels based on different stages. The CPU played as a host to call the GPU kernels in order as a pipeline. Figure 5 shows the pipeline.

During each iteration of the outer most loop, new rays were generated and stored in a ray queue on GPU. In this work, the *Z-order*[1] was used to assign pixels/rays to GPU threads. After new rays were generated, they were all marked as active and would be traced and updated iteratively as follows.

The depth of each active ray path in the queue grew up by one during each iteration, so the active rays always had the same number of bounces. Because some active rays might be absorbed and some might miss all geometries completely after being traced, they were marked as unactive and ignored for the rest of iterations. Then shading work was performed for those rays being active just before the traversal kernel. The direct lighting kernels consisted of two small kernels which were the direct lighting kernel for the light tree and the lighting kernel for directional lights. They were called respectively if both kinds of light sources existed. For the sake of culling active rays for the next iteration, a compaction on the ray queue was performed at the end of each iteration. This was done by using the compaction method in the CUDPP library. What is more, the radix sort method in the CUDPP library was also employed in this work to sort rays for material/texture fetching. The user could define a upper limit on iteration times, representing the maximum number of light bounces considered. If some rays stayed active after the upper limit was reached, they would be discarded.[2]

1 Z-order, Morton-order or Morton code is a space-filling curve used in computer science.

2 This could incur biases in the final image. While the biases were not so obvious if the maximum bounce was not set so small for most diffuse scenarios.

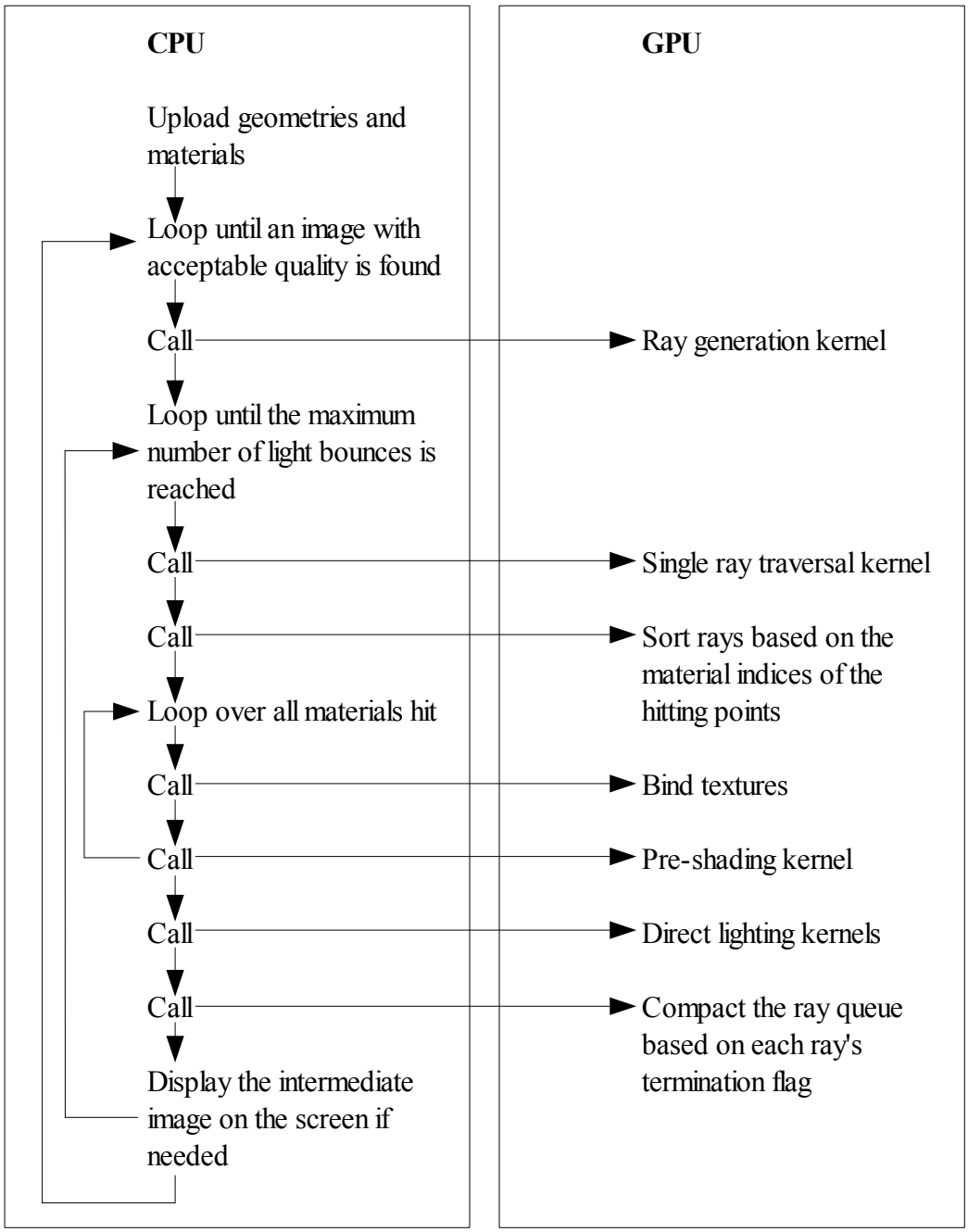


Figure 5: The pipeline of this work.

4 Results and Discussion

4.1 Testing Environments

All tests were performed on an NVIDIA Geforce GTX 260+ GPU with 1.75 GB GDDR3 dedicated memory. The graphics card was installed on a desktop PC with Intel Core i5 quad-core CPU running at 2.67GHz and 8 GB of RAM. The operating system was Windows 7 Professional 64-bit edition. The version of *CUDA* used was 2.3, including the driver and the compiler.

4.2 Results and Discussions

4.2.1 Memory Consumptions

Table 1 shows the specifications of all test scenes and the memory usages on GPU for each of them. As mentioned in section 3.2, geometries were stored as a list of triangles and each of them was in

Scene	Triangles	Point lights	Directional lights	Area lights	BVH size (MB)	Geometry size (MB)	Material size (MB)	Memory footprint (MB)
Cornell Box	34K	0	0	1	0.5	5.9	4.4	212.4
Cathedral	205K	5	0	0	2.9	35.0	162.3	401.9
Sponza	376K	0	1	0	5.1	54.8	10.7	272.4
DMDeck	1804K	49	1	0	25.3	262.2	38.7	528.0

*Table 1: Geometry size referred to the size of the memory space containing all the mesh vertexes, per-vertex normals and UV-coordinates. The memory footprint consisted of all the memories allocated on GPU, including all the materials and textures. In the case of this work, it also contained arrays of per-ray data whose size was resolution-dependent. The resolution of the testings above was 1024*1024 and all the per-ray data under this resolution occupied around 188 MB memory.*

the form of Woop's transformation matrix. Besides the position information, there were also other per-vertex data, such as per-vertex normals and UV coordinates for different material channels. Because the triangles were not duplicated during the tree construction[1], the size of the geometry data grew linearly with respect to the scene's complexity. The size of the BVH also grew linearly

1 In the early split stage, some big triangles were splited and duplicated.

with respect to the complexity of the geometries in the scene. This was because every inner node in the tree had a degree of two and all the triangles existed in the leaf nodes, so that, if the number of leaf nodes was L, then the total number of nodes in the tree was $2 \times L - 1$.

These two conclusions were supported by the figures in the table. Thanks to the compactness feature of BVH, the size of it for a complex scene, such as the DMDeck, was quite small and the size of the geometry information was also acceptable.

Another important part in memory consumption was the per-ray data. For some scenes, it was even the dominant part. But the size of it was constant with respect to the scene's complexity. It scaled linearly with the size of the frame buffer. With the resolution of 512*512, it was about ¼ of the size with 1024*1024.

4.2.2 Performances and Analyses

The average performances for each scene are given in table 2. The figures in the table show the

Max depth of ray paths	Scene	1 shadow ray, 512*512	4 shadow rays, 512*512	1 shadow ray, 1024*1024	4 shadow rays, 1024*1024
5	Cornell Box	11.5 M	22.0 M	17.0 M	29.0 M
	Cathedral	6.5 M	10.5 M	8.5 M	12.5 M
	Sponza	7.5 M	7.2 M	9.0 M	9.0 M
	DMDeck	5.5 M	9.0 M	7.0 M	11.0 M
1	Cornell Box	18.0 M	34.0 M	27.5 M	47.0 M
	Cathedral	11.0 M	15.0 M	16.0 M	19.2 M
	Sponza	12.0 M	12.0 M	18.0 M	18.0 M
	DMDeck	10.0 M	14.0 M	14.5 M	18.3 M

Table 2: The average number of rays per second for each testing scene.

average number of rays per second (including shadow rays) achieved by the path tracer for each test scene. The third and fifth columns show the performances when only one shadow ray was sent at each hit point for light sources in the light tree, while the fourth and sixth columns show the numbers when four shadow rays were sent. It is worth noticing that there was not any light sources in the light tree in the scene Sponza, in which there was only one directional light, such that the number of shadow rays did not make much change on the speed. On the other hand, the performances with a resolution of 512*512 are listed in the third and fourth columns and the ones with 1024*1024 are listed in the last two columns. When the maximum depth of ray paths was one, only primary/eye rays (and the associated shadow rays) were traced. When the depth grew up to five, secondary rays would be traced up to four times for each path.

It is clear that higher resolutions resulted in higher performances. In other words, more rays outperformed less rays. The most important reason for this was that the speed was compromised for

the entire loop which contained many functions and kernels as figure 5 shows. The execution time for some of them did not scale linearly with the number of rays. What is more, the code for the GPU were split into several small kernels to make it flexible and avoid the watchdog[1] problem, but more context switching overheads were incurred at the same time. Those overheads did not depend on the number of rays either. So, launching as many rays as possible at one time could alleviate the impact of the factors above and make the whole process more efficient.

Another conclusion can be made from the table which is that the average speed descended as the ray path grew. The main reason is straightforward that is the coherence between rays could drop more and more as the ray path became longer, as analyzed in [14]. Besides the coherence degradation, the amount of the surviving rays after one reflection due to the absorption probability could become smaller, which also contributed to the inefficient execution of the loop as discussed in the last paragraph.

How to make the tracing of secondary rays efficient is a non-trivial problem, [13][14][15][18] have discussed/analyzed the performances of tracing secondary rays. While, if the problem became how to get a better result within a shorter time, there could be another alternative which was to sample more light sources and trace more shadow rays.

Tracing shadow rays was much cheaper than tracing secondary rays in this work. The reason came from two aspects. The first one was that shadow rays were just used for checking visibilities and a shadow ray could return right after finding an intersection in a BVH. While, for primary and secondary rays, the nearest intersection points should be returned so that rays could only leave the kernel when the traversal stacks were empty. Another aspect was that when tracing more than one shadow ray for each hit point, shadow rays could be more coherent than secondary rays because they could share the same origin and head for similar directions. The percentages of time consumed for each kernel on GPU for three scenes are shown below.

Scenes	Kernels	Depth=1,shadow=1	Depth=1,shadow=4	Depth=5,shadow=1	Depth=5,shadow=4
Cornell Box	Traversal kernel	39.8%	25.2%	48.2%	31.9%
	Lighting kernel	24.6%	52.2%	26.1%	51.1%
	Others	35.7%	22.6%	25.7%	17.0%
Cathedral	Traversal kernel	27.9%	12.7%	47.0%	27.0%
	Lighting kernel	49.0%	76.8%	34.2%	62.1%
	Others	23.1%	10.5%	18.8%	10.9%
DM-Deck	Traversal kernel	26.3%	14.0%	44.7%	29.4%
	Lighting kernel	36.9%	66.6%	26.1%	51.2%
	Others	36.8%	19.4%	29.2%	19.4%

Table 3: The percentages of time of GPU kernels for 3 scenes. Profiled by CUDA Visual Profiler.

1 There is a watchdog timer in operating systems with a desktop. If a GPU process does not return within a time limit (a system dependent value, typically, about several seconds), the watchdog will enforce the GPU process to return without caring if the task is finished.

The relative cost of a shadow ray compared to a primary (and secondary) ray can be found by dividing the percentages of the lighting kernel by the percentages of the traversal kernel and the number of shadow rays. The result shows in figure 6. It verifies the conclusions in the last paragraph as the relative cost of a shadow ray went down both when secondary rays were taken into account and more shadow rays were traced at each point.

To measure the actual speed of convergence of the images rendered with different settings, the variance of an image was defined as the average for each pixel of the sum of the squared difference of each RGB channel to a reference image's. The expression was

$$Variance = \frac{1}{M}\sum_{i=0}^{M}(pixel[i].r - ref[i].r)^2 + (pixel[i].g - ref[i].g)^2 + (pixel[i].b - ref[i].b)^2,$$

where M was the total number of pixels in the image.

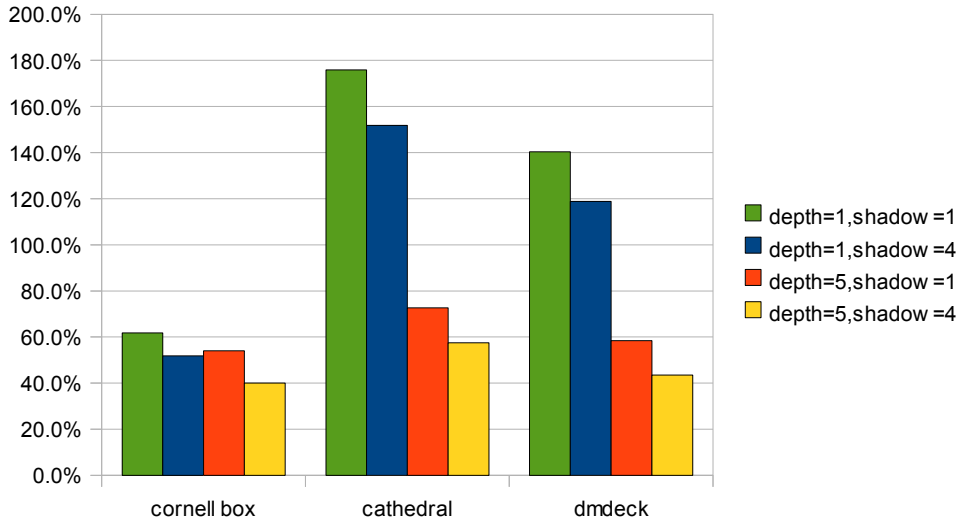


Figure 6: The relative cost of a shadow ray in 3 scenes.

The irradiance map was used in this test to get rid of the potential affect of the surface materials. The reference image was rendered for four hours. They are shown in figure 9, and there was almost no noise left in these images. Because there was only one directional light in the scene Sponza, this test was not relevant.

The variances of the rendered images were checked during the first twelve minutes. Figure 7 shows the change curve of the variance based on the time and figure 8 shows the change curve based on the total number of rays traced. The comparison images are listed in figure 10.

Figure 7 reflects that, except the scene Cornell Box, tracing more shadow rays made the convergence faster in time in the first twelve minutes. While figure 8 reflects that, except the image Cathedral 1, a lower ratio of shadow rays could reduce the variance if the total number of rays traced were fixed.

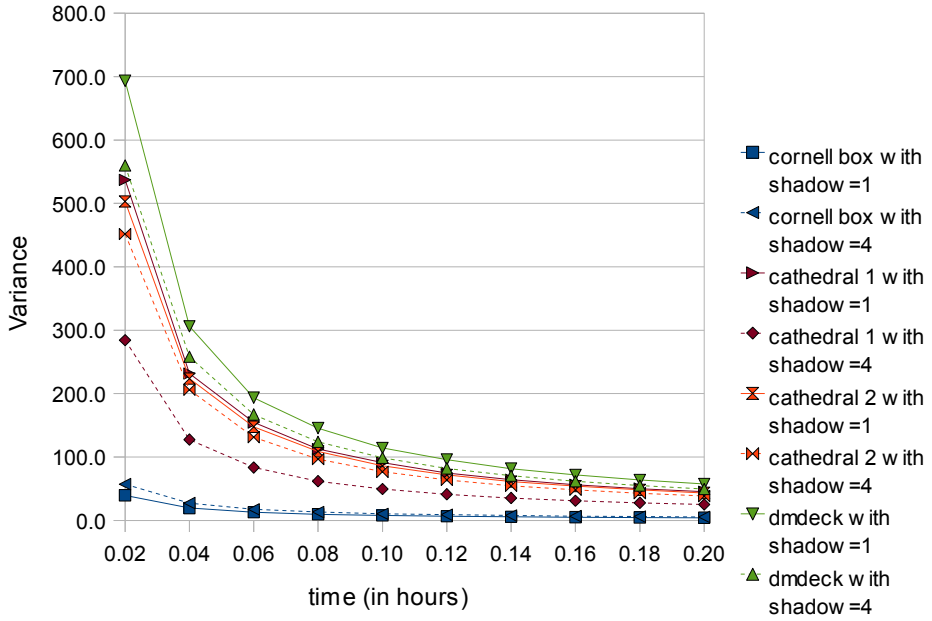


Figure 7: Comparison of convergence based on time elapsed.

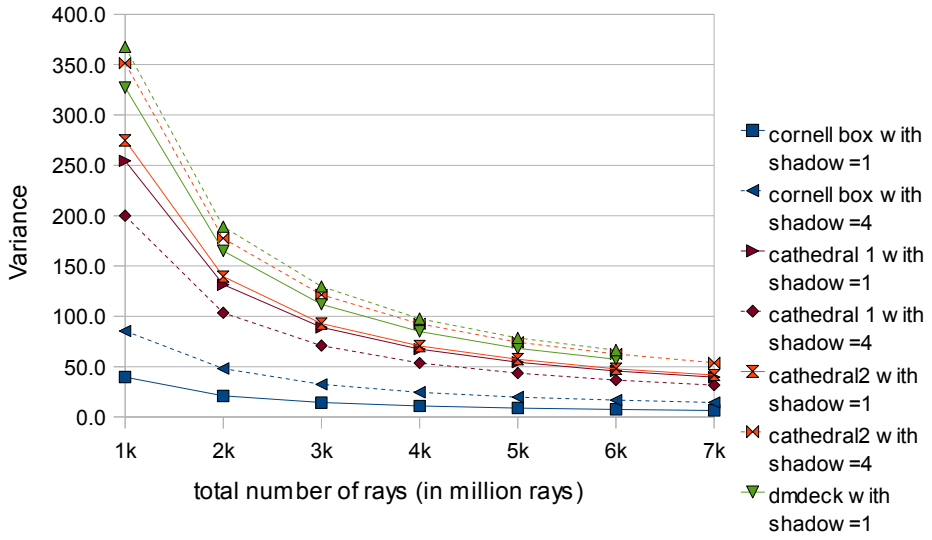
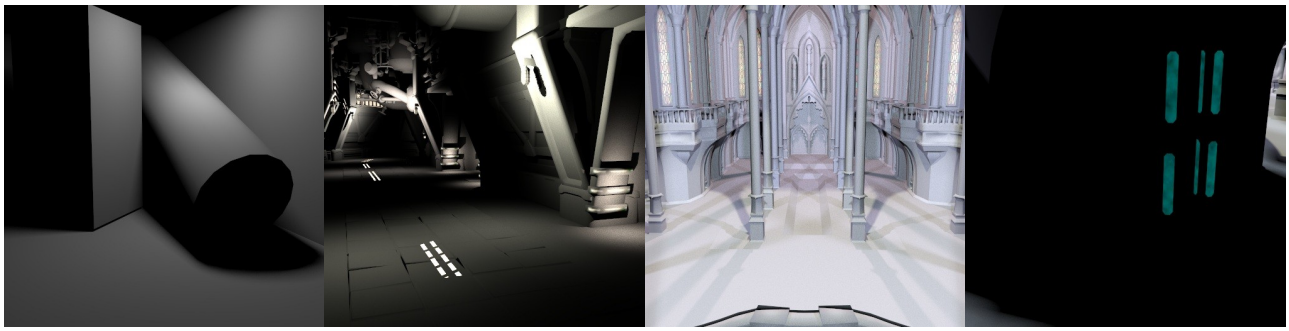


Figure 8: Comparison of convergence based on total number of rays traced.

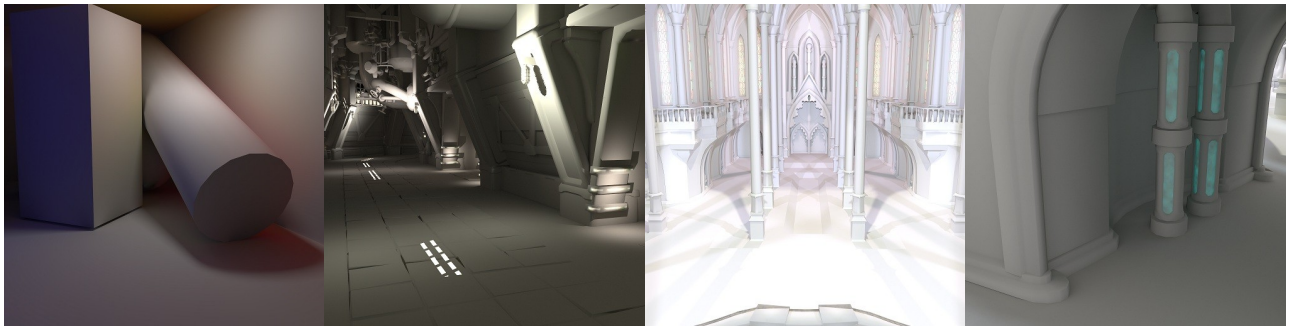
Reasonable explainations for these were discussed as follows. For the image of Cathedral 1, most pixels were dominated by more than one light source directly, so that, in either case, sampling more light sources/tracing more shadow rays could bring more benefits. While the image of Cathedral 2 was a comparision because most of the pixels were lit only by indirect light. The conditions for the images of Cornell Box and DMDeck were similar, where quite a few pixels were also dominated by indirect light. So tracing more secondary rays reduced variance more efficiently in these cases. Since the lighting setups for the scene DMDeck and Cathedral were complex and many parts were

affected by more than one light source, increasing the number of light samples/shadow rays still accelerated the convergence of the image due to the efficiency of tracing shadow rays. But in the case of the Cornell Box, only soft shadow edges could be affected by multiple light samples effectively so that tracing more than one shadow ray was just wastful in general.

More images are listed in figure 11.

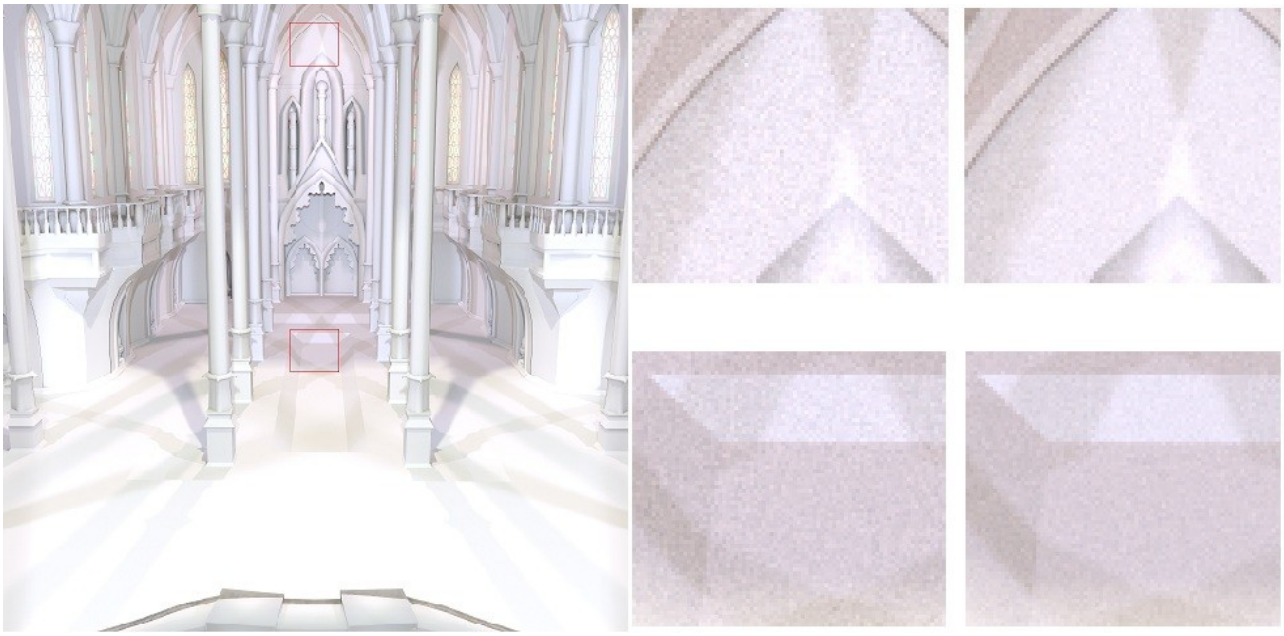


Images rendered with only direct lights.

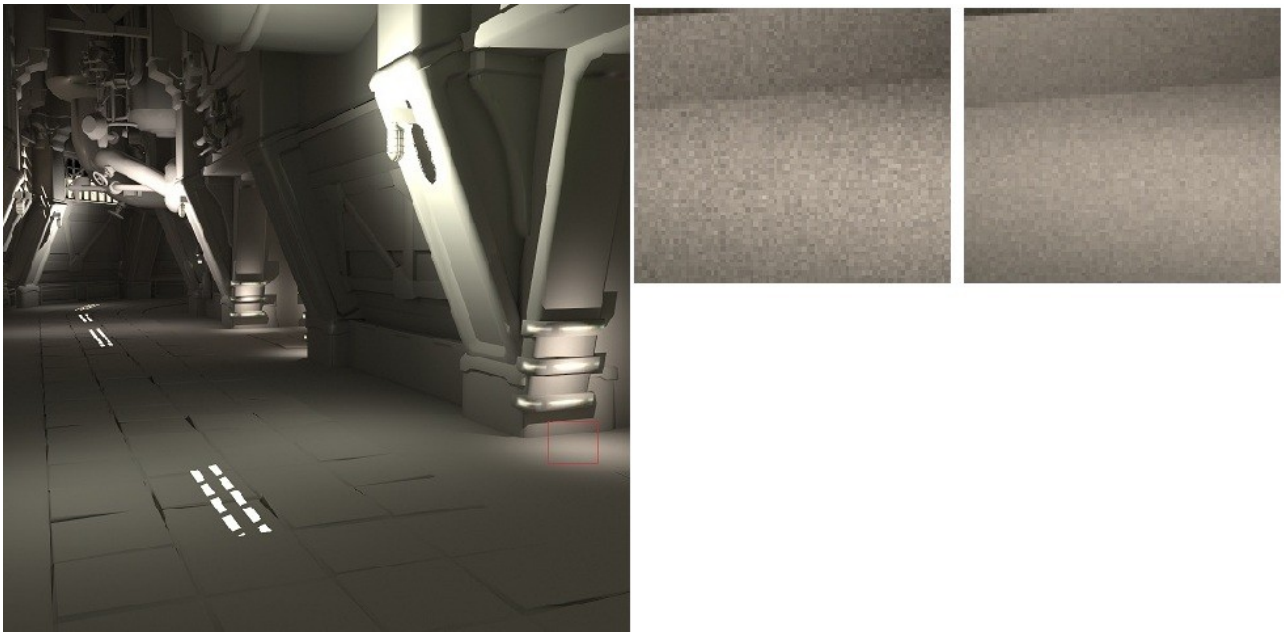


Images rendered with both direct and indirect lights (max depth=5). These four images were rendered over 4 hours and there was almost no noise left. They were used as the reference images for the variance tests.

Figure 9: The irradiance maps of Cornell Box, DMDeck, Cathedral 1 and Cathedral 2 (from left to right).

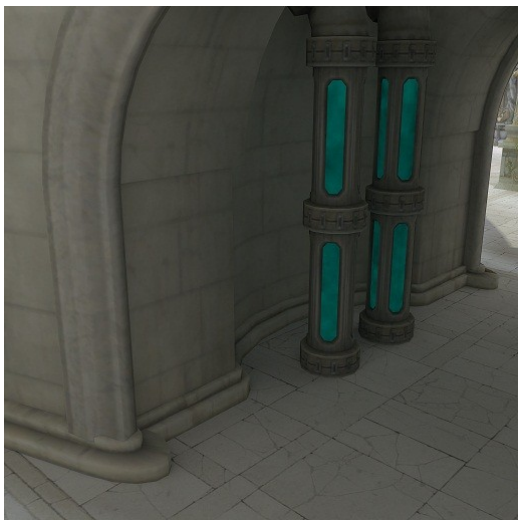


Cathedral 1



DMDeck

Figure 10: The detailed comparisions of convergence. The small images on the left were rendered with 1 shadow ray per step for 12 minutes while the ones on the right were rendered with 4 shadow rays for the same length of time.

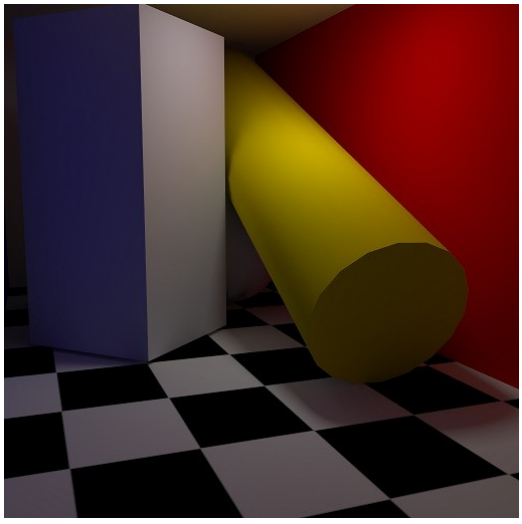


Cathedral

Cathedral

DMDeck

DMDeck



Cornell Box

Sponza

Figure 11: Images rendered by the GPU path tracer in this work.

5 Conclusion and Future Work

5.1 Conclusion

The path tracing algorithm was implemented on GPU successfully by using *CUDA* in this work. The BVH was utilized as the acceleration structure. A light tree structure was also introduced and implemented so that it could cull light souces with an affecting radius efficiently. By testing and analyzing the performances and memory consumptions for different scenes, the implementation in this work proved to be efficient and stable enough to render real game levels with complex lighting setups, and it was also compact in memory consumption such that real game levels could be fit into a modern graphics card.

Three texture types and three light source types were supported in this work as well. By introducing the pipelined structure in section 3.5, the entire process was split into seperate kernels. In this case, kernels could be modified easily to extend for more features and it was easy to debug. What is more, the actual speed of convergence was measured and analyzed for different scenes and the inference to config the path tracer was got as follows. If there were many light sources in the scene and many of them had its affecting region overlapped with others', sampling more light sources and tracing more shadow rays could render an image with less noice faster.

5.2 Future Work

For the next step, more texture types, such as transparent textures, and more light sources types, such as sky lights, need to be supported. If the total number of supported types are big, the resulting kernels can be very branchy and large. Currently, the shading work was far from becoming the bottleneck. But if the code for shading becomes very complex and the execution time becomes significant, moving the full shading work to CPU may be a better choice.

Instancing can be supported as well and it will make the memory for storing the geometries much smaller. Besides, the memory taken by the ray queue can also be made smaller. To make the memory consumption even more compact, materials can be offloaded to CPU and corresponding data can be streamed to a common place on GPU only when necessary.

Additionally, supporting multiple GPUs is a quite useful extension. The simpliest solution is to split the rendering screen equally and keep a copy of all the data on each GPU. It is easy to implement and efficient to execute while the drawback is the memory used for each GPU is not compromised because of the bigger amount of memory available in total. A more advanced solution is to make multiple GPUs share both workload and memory consumption.

A hybrid solution of both CPU and GPU can also be promising.

6 References

1. Dutré, Philip, Philippe Bekaert and Kavita Bala. *Advanced Global Illumination*, page 1-6, 54-63, 68-69, 128. Natick, MA: A K Peters, 2003.
2. Akenine-Möller, Tomas, Eric Haines and Naty Hoffman. *Real-Time Rendering, Third Edition,* page 417-425. Wellesley, MA: A K Peters, 2008.
3. CRYTEK. *CryENGINE®3: Next Generation Real-Time Graphics* [online]. Available: http://www.crytek.com/technology/cryengine-3/specifications/ [accessed 17 May 2010].
4. Dachsbacher, Carsten and Marc Stamminger. Reflective Shadow Maps. In *Proc. of the Symposium on Interactive 3D Graphics and Games*, page 203-213, 2005.
5. Ritschel, T., T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher and J. Kautz. Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph.*, 27(5):1-8, 2008.
6. Purcell, T. J., I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Trans. Graph.*, 21(3):703-712, 2002.
7. Sloan, P-P., J. Kautz, and J. Snyder. Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments. *ACM Trans. Graph.*, 21(3):527-536, 2002.
8. Nvidia Corporation. *NVIDIA CUDA Programming Guide, Version 2.2*. 2009.
9. Nvidia Corporation. *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Version 1.1*. 2009.
10. Popov, Stefan, Johannes Günther, Hans-Peter Seidel and Philipp Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *EUROGRAPHICS*, 26(3):415-424,2007.
11. Günther, Johannes, Stefan Popov, Hans-Peter Seidel and Philipp Slusallek. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Proc. IEEE/Eurographics Symposium on Interactive Ray Tracing*, page 113-118, 2007.
12. Zhou, Kun, Qiming Hou, Rui Wang and Baining Guo. Real-Time KD-Tree Construction on Graphics Hardware. *ACM Trans. Graph.*, 27(5):1-11, 2008.
13. Boulos, S., D. Edwards, J. D. Lacewell, J. Kniss, J. Kautz, P. Shirley, and I. Wald. Packet-based Whitted and Distribution Ray Tracing. In *Graphics Interface*, page 177–184, 2007.
14. Månsson, Erik, Jacob Munkberg and Tomas Akenine-Möller. Deep Coherent Ray Tracing. In *Proc. IEEE/Eurographics Symposium on Interactive Ray Tracing*, page 79-85, 2007.
15. Aila, Timo and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of High-Performance Graphics*, page 145-150. New York, NY: ACM, 2009.

16. Wang, Rui, Kun Zhou, Minghao Pan and Hujun Bao. An Efficient GPU-based Approach for Interactive Global Illumination. *ACM Trans. Graph.*, 28(3), 2009.

17. Hachisuka, Toshiya, Shinji Ogaki and Henrik Wann Jensen. Progressive Photon Mapping. *ACM Trans. Graph.*, 27(5), 2008.

18. Zlatuška, Martin and Vlastimil Havran. Ray Tracing on a GPU with *CUDA* – Comparative Study of Three Algorithms. In *WSCG 2010 conference, Communication Papers Proceedings*, page 69-76, 2010.

19. Ernst, Manfred and Günther Greiner. Early Split Clipping for Bounding Volume Hierarchies. In *Proc. IEEE/Eurographics Symposium on Interactive Ray Tracing*, page 73-78, 2007.

20. Wald, Ingo. On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proc. IEEE/Eurographics Symposium on Interactive Ray Tracing*, page 33-40, 2007.

21. Woop, Sven. A Ray Tracing Hardware Architecture for Dynamic Scenes. Tech. Rep., Saarland University, 2004.

Appendix A

A.1 Terms and the Rendering Equation

A.1.1 Radiance

Radiant power, or flux, is the measurement for the energy of light. It is often denoted as Φ whose dimension is Watt. This quantity expresses how much energy flows through a surface per unit time. The quantity irradiance and radiosity describe the incident/exitant radiant power per unit surface area.

$$E = B = \frac{d\Phi}{dA}$$

Radiance can be used to measure the energy along a certain direction. It describes the flux per unit area per unit solid angle.

$$L = \frac{d^2\Phi}{d\omega\, dA\cos\theta}$$

This notation: $L(x \rightarrow \Theta)$ represents the radiance outgoing from a point x along direction Θ and $L(x \leftarrow \Theta)$ represents the receiving radiance. Additionally, radiance has an important property that it stays constant along a straight path.

$$L(x \rightarrow y) = L(y \leftarrow x)$$

A.1.2 BRDF

BRDF, which is the abbreviation of bidirectional reflectance distribution function, describes the surface properties of an object and the reactions of light that hit that surface. It is defined as the ratio of the differential radiance reflected in an exitant direction Θ, and the differential irradiance incident through a differential solid angle Ψ at a certain point x. The BRDF is denoted as:

$$f_r(x, \Psi \rightarrow \Theta) = \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \Psi)}$$
$$= \frac{dL(x \rightarrow \Theta)}{L(x \leftarrow \Psi)\cos(N_x, \Psi)\, d\omega_\Psi}.$$

Specifically, the *Phong* model is used in this work, which has the form of

$$f_r(x, \Psi \rightarrow \Theta) = k_s \frac{(R \cdot \Theta)^n}{N \cdot \Psi} + k_d .$$

A.1.3 The Rendering Equation

If $L_e(x \rightarrow \Theta)$ represents the radiance emitted from point x in direction Θ and $L_r(x \rightarrow \Theta)$ represents the radiance that is reflected at that point in the direction Θ, the following equation holds.

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta) \quad (1)$$

By introducing the definition of BRDF, the following equations can be deduced.

$$L_r(x \rightarrow \Theta) = \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta) L(x \leftarrow \Psi) \cos(N_x, \Psi) d\omega_\Psi$$

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_{\Omega_x} f_r(x, \Psi \rightarrow \Theta) L(x \leftarrow \Psi) \cos(N_x, \Psi) d\omega_\Psi \quad (2)$$

Equation 2 is one form of the rendering equation. It performs a hemispherical integral for every visible point. Mathematically, it can be transformed into another form which integrates over all the surface areas in the scene for every visible point. The new equation has the form of

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_A f_r(x, \Psi \rightarrow \Theta) L(y \rightarrow -\Psi) V(x, y) \frac{\cos(N_x, \Psi) \cos(N_y, -\Psi)}{r_{xy}^2} dA_y \ (3),$$

where $V(x,y)$ stands for the visibility function. The term $G(x,y)$ presents the geometry relationship between surface point x and y, having the following form:

$$G(x, y) = \frac{\cos(N_x, \Psi) \cos(N_y, -\Psi)}{r_{xy}^2} \quad (4).$$

Then equation 3 can be reformulated as:

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_A f_r(x, \Psi \rightarrow \Theta) L(y \rightarrow -\Psi) V(x, y) G(x, y) dA_y \quad (5),$$

by introducing equation 4.

A.2 Monte Carlo Methods

Monte Carlo Methods are based on the statistics theory that the frequency gets close to the real probability asymptotically as the number of samples grow. So if there is a variable called estimator, which has the form

$$S = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \quad (6),$$

where $p(x_i)$ is the probability density of x_i , then the expectation of this estimator is

$$
\begin{aligned}
E[S] &= E[\frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}] \\
&= \frac{1}{N}\sum_{i=1}^{N}E[\frac{f(x_i)}{p(x_i)}] \\
&= \frac{1}{N}N\int\frac{f(x)}{p(x)}p(x)\,dx \\
&= \int f(x)\,dx \\
&= I\ .
\end{aligned}
$$

The variance of the estimator S is

$$\sigma^2 = \frac{1}{N}\int(\frac{f(x)}{p(x)} - I)^2\,p(x)\,dx \quad (7).$$

The Monte Carlo integration technique can be extended to a multi-dimensional form in a straightforward way. The form of two dimensions shows as follows:

$$
\begin{aligned}
I &= \int\int f(x,y)\,dxdy\ , \\
S &= \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i,y_i)}{p(x_i,y_i)}\ .
\end{aligned}
$$

The variance can be rewritten as, by extending the square term as in equation *7*:

$$\sigma^2 = \frac{1}{N}(\int\int\frac{f(x,y)^2}{p(x,y)}dxdy - I^2) \quad (8).$$